computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

THE ART AND SCIENCE OF GPU AND MULTI-CORE PROGRAMMING

Robert E. Hiromoto

University of Idaho, 1776 Science Center Drive, Idaho 83402, USA,
hiromoto@cs.uidaho.edu

Abstract: *This paper examines the computational programming issues that arise from the introduction of GPUs and multi-core computer systems. The discussions and analyses examine the implication of two principles (spatial and temporal locality) that provide useful metrics to guide programmers in designing and implementing efficient sequential and parallel application programs. Spatial and temporal locality represents a science of information flow and is relevant in the development of highly efficient computational programs. The art of high performance programming is to take combinations of these principles and unravel the bottlenecks and latencies associate with the architecture for each manufacturer computer system, and develop appropriate coding and/or task scheduling schemes to mitigate or eliminate these latencies.*

Keywords: *Performance, Speedup, Parallelism, Temporal Spatial Locality.*

1. INTRODUCTION

Commodity multithreaded processors in dual- and multi-core processors, and the graphic processing unit (GPU) comprise today's computer market place. The progress in the nanometer process technology has advanced the design and manufacturing of these ambitious computer architectures. This new era of computing capabilities place new demands on the programmers and requires potentially new tricks or programming techniques for veteran programmers experienced in the art of parallel programming. With each new generation of more sophisticated processor design, the processor structure, the organization of memory hierarchy, the implicit and explicit scheduling of executable threads (tasks), and the programming software environment reappear as a set of new criteria that defines the parameter space of programming paradigms. Encouraged by the promise that these new systems can deliver higher performance, a new surge of interest and activity has emerged in areas of multi-core and GPU computing. As a consequence, the GPU is now manufactured with higher precision arithmetic and programming software to allow general-purpose computing on GPUs (GPGPU), where traditionally the applications were handled by the CPU. The GPGPU programming paradigm relies on streaming data through lightweight threads of execution, where banked memory provides non-blocking access of operands (data) between competing threads. Consequently, hardware support for GPUs and multi-core systems require more complex memory hierarchies, and hardware technologies to maintain memory consistency and cache coherence under a dynamic parallel execution model. Cache coherence in multicore systems is an example of the hardware organizational complexity required to address memory latencies.

New technologies, like the GPU and multi-core processor, introduce new challenges to the applications programmer. The novelty of these systems lies in their hardware organization, where an optimal mapping between hardware and software can easily be compromised when multithreaded parallel execution is introduced. The challenges for a parallel code (program) developer is to bend, reshape, and retrofit existing codes onto new and ever changing architectures. For the novice developer, this requires navigating a desperate terrain strewn with hidden and undetected detours.

Fortunately, there are physical principles that can provide code developers with a framework to reason about multithreaded programming complexities exhibited by new and intricate hardware configurations. These principles are *time* and *space* metrics upon which performance is measured. The faster the clock cycle, the higher the potential CPU performance; or the longer in response (latency) time, the lower in performance that can be expected. The rate of *information* flow, ultimately dictates the achievable performance whether in the activities of the financial markets or the execution of applications on computer systems. Minimizing the time for operand access, repeated reuse of data or instructions, minimizing latencies between input/output (I/O) requests are examples where *temporal* and the *spatial* locality between references play important roles in optimizing performance. *Temporal* and *spatial* localities are two simple principles that all programmers, interested in designing and implementing optimally performing application codes, should be guided by.

In the next sections, we introduce the concept of *spatial* and *temporal* locality and the implied consequences to the organization of memory hierarchies and coding of the matrix multiplication program; the notion of parallelism and Amdahl's law; and architectures of the multi-core and the GPU.

2. SPATIAL AND TEMPORAL LOCALITY

The principle of locality [1, 2] refers to two basic types of reference locality. *Spatial* locality refers to the use of data elements stored within a relatively close proximity of one another. *Temporal* locality refers to the reuse of data (instructions) or other resources within relatively short time periods. These principles, in various forms, are applied in performance optimization for cache utilization and memory prefetching technology, code motion affecting memory access patterns, and process scoreboarding to enhance processor performance. Although there are other specific terminologies of locality, those more dynamic predictive assertions (branch or most probable access predictor) are typically beyond the control of the programmer.

In this paper, we take the liberty to expand the definitions of both *spatial* and *temporal* locality as measures of both near and distant proximity of data references. Under these expanded definitions, *spatial* and *temporal* localities are principles that encapsulate the basic notions of length and time. They are measurable and when taken separately or in combinations, they can be reasoned about and formulated in predictive analysis.

From a programmer's perspective, such metrics lend themselves in code planning for I/O tasks and data layouts, code restructuring for both sequential and parallel execution, and reasoning within the context of a memory hierarchy with associated referencing costs. The programmer must understand the hardware organization and resources available to the application code. As with every organizational structure, computer system has their particular idiosyncrasies; however, these peculiarities can be measured in both space and time.

2.1. MEMORY HIERARCHY

Hierarchical memory is a hardware organization that is arranged to benefit from *spatial* and *temporal* references. Levels of memory characterize a typical memory hierarchy; where caches, populated near the CPU, have low latency and low memory capacity devices. As the distance between the CPU and the levels (from low to high) of the memory hierarchy increase, so to do their access latencies and memory capacities increase. As the distance between the CPU and the levels (from low to high) of the memory hierarchy increase, so to do their access latencies and memory capacities increase. The design of hierarchical memory is organized to read blocks of slower (higher) level memory into the next successive (lower) level of the memory hierarchy. Predictive algorithms (hardware and OS) attempt to predict which data might be accessed next or least and then react by moving the appropriate data through the memory hierarchy as needed. Temporal latencies and memory device capacities are illustrated in Fig. 1, where the cost of accessing data from the memory hierarchy that are more distant from the CPU increases from a few clock cycles to orders of magnitude.

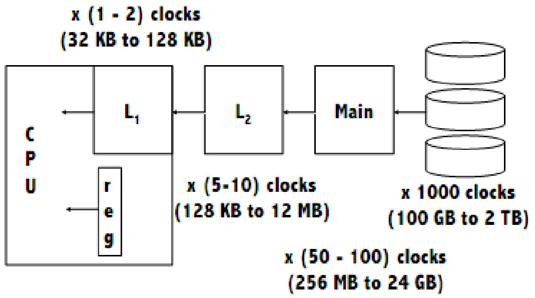
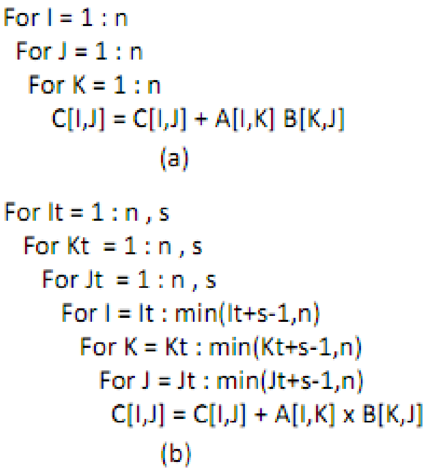


Fig. 1 – Memory Hierarchy

The organization of a memory hierarchy illustrates a principle of economy of scale. A programmer should be aware of this principle when dealing with I/O considerations. A cache is designed to be faster by restricting operand address lookup to within a small memory capacity, and by requiring shorter wires (links) to connect to the CPU. In exchange for limited cache capacity, cache technology employs temporal and *spatial* locality analyses to maintain recently referenced data and/or near recently referenced data. Programmers have considerable control over the locality of data referencing by structuring their codes in ways to increase the *spatial* and *temporal* locality of both operand and instruction references, and the locality offered by the various hierarchical storage devices. The result of this analysis can lead to a higher cache utilization per unit cycle time; translating into higher performance.

2.2. MATRIX MULTIPLICATION: ANALYSIS

An interesting application of spatial and temporal locality is the coding of the matrix multiplication operation [3]. In most introductory lectures, the multiplication of two matrices is commonly defined through the loop structure as illustrated in Fig. 2 (a). The inner-loop index K references elements of matrix A in a row-wise manner (in cache), whereas, the elements of matrix B are referenced column-wise. In a programming language that assigns the elements of a matrix in a row-wise fashion, the cache strategy for predicting the most probable next reference will load the elements of matrix B in a row-wise fashion as well. The affect of program (a) and the caching predictive strategy results in an inefficient cache utilization for matrix B, where fewer elements of its column-wise references are available in the cache resulting in more cache misses; as compared to cache misses experienced by row-wise references of matrix A. In this example, a simple remedy requires only an interchange between the J and K loop indexes. In this reorganized loop structure, matrix B more optimally complies with maximizing *spatial* locality, whereas, the matrix element A[I,K] now exhibits *temporal* locality. Fig. 2(b) illustrates a blocking technique that further improves *temporal* and *spatial* locality. In this more complicated set of loops, matrix multiplication is computed using a series of sub-block calculations where sub-blocks of A and B can be used several times (increasing *temporal* and *spatial* locality) before the next sub-blocks are accessed by the cache in a block consecutive fashion.



```
For I = 1 : n
 For J = 1 : n
  For K = 1 : n
    C[I,J] = C[I,J] + A[I,K] B[K,J]
```

(a)

```
For It = 1 : n , s
  For Kt  = 1 : n , s
    For Jt  = 1 : n , s
      For I = It : min(It+s-1,n)
        For K = Kt : min(Kt+s-1,n)
          For J = Jt : min(Jt+s-1,n)
            C[I,J] = C[I,J] + A[I,K] x B[K,J]
```

(b)

Fig. 2 – Titled Matrix Multiplication (See [3])

A simple space-time analysis of program (a) resulted in a more optimal sequential implementation of the matrix multiplication operation. Yet a more surprising result of program (b) is that it also represents a parallel formulation of matrix multiplication. This is a very rich result as no parallel programming considerations are explicitly employed. Guided only by the analyses of spatial and temporal locality, a rather deep and unexpected result has emerged. As a consequence, it is challenging to speculate on the applicability of spatial and temporal locality as an analysis tool in developing more optimal sequential and parallel (not necessary more parallel but more optimal in performance) algorithms. The notion here is to provide distance and time metrics to determine the rate or radius of information that influences the final solution. Once such an algorithmic analysis is completed the details of computer implementation then follow. Program (b) represents a blocking strategy that parallel algorithms employ in certain situations of their implementation. The GPU in particular is heavily reliant on this technique termed *tiled* memory to facilitate memory accesses and kernel executions.

3. CATEGORIES OF PARALLELISM

Parallel programming is an optimization technique to increase computational performance. The idea is as old as human and pre-human activities in scavenging for food, where independent journeys are made to search and returned to their home base with their share of findings. Computational parallelism incorporates a similar strategy. The solution of a single problem is divided up into as many independent tasks (i.e., no information flows between these tasks) that can then be executed concurrently and then accumulated to obtain the final solution. Parallelism may be categorized into two broad groups: (1) unbounded and (2) bounded parallelism. Unbounded (or near unbounded) parallelism requires little or no communication (transfer of information) between any of the independently executing tasks. The term "embarrassingly parallel" is coined to describe such parallelism. Unbounded parallelism ushered in the notions of scalability and scaled speedup [4] as desired characteristics for trends in the future development of parallel algorithms. Unbounded parallelism may be attained in searches, algorithms that use data (information) inefficiently, transforming or marshaling data, rendering in computer graphics, ray tracing, or brute-force searches in cryptography to name a few examples. Bounded parallelism represents a more typical class of constrained parallelism, where the solution specifications requires frequent or nominal exchange of solution critical data that are computed incrementally among the various parallel tasks. All real-world solution-oriented processes represent this behavior. Fig. 3 illustrates this point in an analogy of the well-known space-time diagram to the space-time division between unbounded and bounded parallelism. Within the space-time cone of influence, solution algorithms are governed by the exchange of solution-oriented information. All communication is bounded by the speed of communication (*C*). Outside the cone of influence, a region of unbounded parallelism reigns; however, for this parallelism to make a solution contribution their region must collapse into the cone of influence.

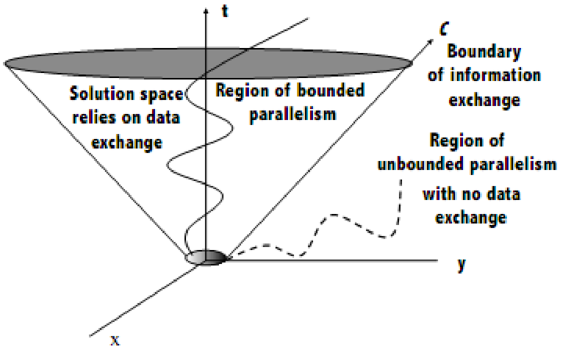

t
C
Boundary of information exchange
Solution space relies on data exchange
Region of bounded parallelism
Region of unbounded parallelism with no data exchange
y
x

Fig. 3 – Regions of Bounded and Unbounded Parallelism

3.1. AMDAHL'S LAW

Amdahl's law [5] is an algebraic argument that parallelism, for a fixed size problem, is limited by the percentage of code that cannot be parallelized. The argument provides a simple but profound insight into the limits of performance that is attainable through the application of parallel techniques and hardware. Amdahl assumes a simple model by neglecting potential bottlenecks such as memory bandwidth and I/O bandwidth. Still the results provide an ideal upper bound for performance expectations. Amdahl's analysis derived the following overall speedup improvement when p processors are applied to the portion of a sequential algorithm that supports f_p percent of parallel improvement.

$$S_p = \frac{1}{f_s + \frac{f_p}{p}}$$

where f_s is the percent of scalar (sequential) code and the condition that $f_p + f_p = 1$.

Speedup is a useful metric to assess performance improvement; however, it is important to realize that it may be misleading as a performance metric since performance is a metric measured in time. Equation (1) arises from the ratio between the sequential execution time of an algorithm over its parallelized execution time. As a result, speedup is a dimensionless improvement metric, independent of time.

Fig. 4 depicts the comparison of speedups between two different algorithms constructed to exhibit high and low percentages of parallelism. Algorithm 1 is constructed to exhibit 90% parallelism, by selecting values for f_s and f_p to be 0.1 and 0.9, respectively. Algorithm 2 is constructed to exhibit low parallelism at 30%. The corresponding values for f_s and f_p are chosen at 0.7 and 0.3, respectively. However, the single processor performance of algorithm 2 is adjusted to be five (5) times the performance of algorithm 1. As might be expected, algorithm 2 never surpasses the speedup of algorithm 1. On the other hand, Fig. 5 provides a more sobering assessment of performance and emphases the pitfalls of the speedup metric. In this comparative plot, algorithm 1 fails to attain a parallel performance faster than algorithm 2, no matter the number of parallel processors employed. The constructed algorithms 1 and 2 could very well represent two different algorithms for the solution of the same computational problem; e.g., variants of the Jacobi and Gauss-Seidel iteration schemes. In this context, more parallelism does not necessarily mean better performance.

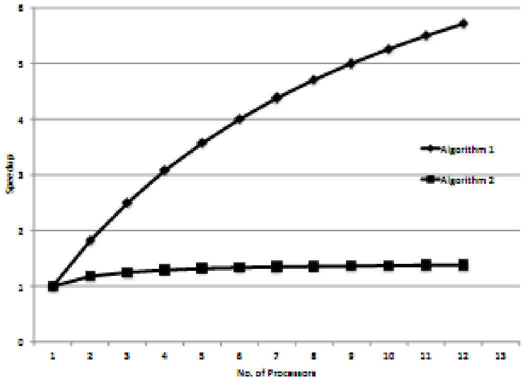


Fig. 4 – Speedup Improvement

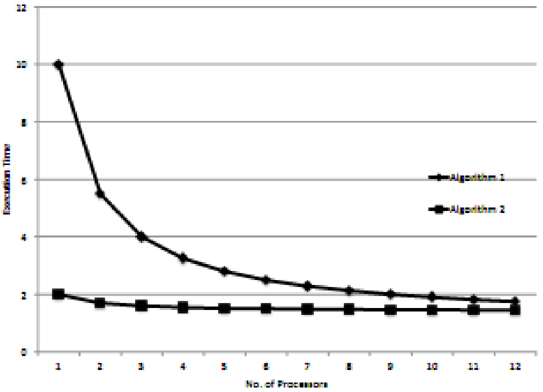


Fig. 5 – Parallel Performance

The problem with the use of speedup as a metric for performance is the introduction of time-normalization based on a non-optimal sequential algorithm (algorithm 1). Fortunately, speedup can be salvaged if the time-normalization is made with respect to the best available sequential algorithm [6], in this case algorithm 2. In other words, the time base applied to the metric of speedup must to explicitly stated; otherwise, the use of the metric is meaningless. Time, as mentioned earlier, is the critical metric that describes performance.

4. GPU AND MULTI-CORE ARCHITECTURES

In this section, an over review of generic architectures for both the multi-core and GPU processors are provided. Saline issues of the architecture are presented and the challenges that they pose for programmers to overcome.

4.1. MULTI-CORES

Traditional processors are designed with only a single CPU that performs the reading and executing of instructions. A multi-core processor combines two or more cores (CPUs) on a single integrated circuit (IC). It is also known as a chip multiprocessor (CMP), and offers the promise of enhanced performance for the processing of independent tasks, a reduced power consumption (by lowering the clock speed on each core), and low-level hardware support for the synchronization of parallel tasks. Ideally, a dual-core processor should have twice the performance as a single core processor. In several benchmarks, performance gains are found to be in the range of fifty percent to one-and-a-half times as powerful as a single core processor [7].

Many-core refers to many cores in a computer. They may or may not all populate the same chip. They may be organized as many single-core chips, a collection of single-core and multi-core chip, or many multi-core chips. A typically “many-core” might refer to 32 or more cores, while “multi-core” are fewer in number. These terminologies are technology driven and certain to change over time.

Fig. 6. depicts a generic dual-core processor with CPU-level 1 (L1) caches and a shared level 2 (L2) cache. A dual- (multi-) core configuration is distinguished from traditional multiprocessor systems by the tight shared-coupling of the L2 cache between two cores. Although this provides a “faster” connection between the coupled cores, it introduces a departure from the hierarchical memory strategies previously discussed. The capability of high-speed accessing of the L2 cache by shared cores presents challenges absent in traditional single-core multiprocessor systems. The multi-core organization places more demands on the predictive data access strategies applied to the shared L2 cache. This demand has the potential to increase reference latencies and increase contention between cores in supplying instructions and data; significantly degrading performance [8, 9]. As a consequence, multi-core designers have provided multiple hardware contexts [10, 11] or multithreading. Multithreading has the advantage that it can handle arbitrarily complex access patterns even in cases where it is impossible to predict the accesses before hand. Multithreading simply reacts to cache misses that occur, rather than attempting to predict them. Multithreading tolerates latency by attempting to overlap the latency of one context with the computation of other concurrent contexts. As with all advantages, there are disadvantages: (i) multithreading relies on available concurrency within an application, which may not exist; (ii) an overhead is incurred in switching between contexts; and (iii) significant hardware support is required to minimize context-switching overhead delays.

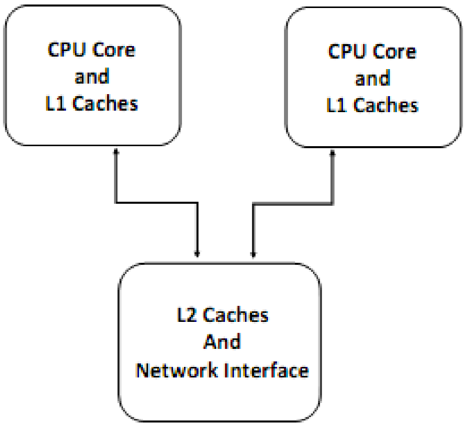


Fig. 6 – Generic Dual-core Processor

In support of multithreading, on-chip synchronization primitives are provided in hardware and enables low synchronization algorithm capabilities [12,13,14]. Low synchronization algorithms offer relaxed ordering constraints on memory operations and reduce or eliminate the synchronization overheads from locking (lock-free) [15]. Ultimately, low synchronization algorithms can improve efficiency and scalability in multithreading. The need for on-chip inter-core communication to handle issues like synchronization and data sharing are being addressed by the developments in Network-on-Chip (NOC) [16,17]. The NOC network links (wires) are shared by many signals. This capability allows NOC to operate on different data packets at one time, achieving a high level of data parallelism, and providing separation between computation and communication on a multi-core processor system. As a consequence, NOC provides high throughput performance and scalability when compared to point-to-point or shared bus communication architectures. The benefits of NOC to multi-core processors could be considerable.

As multi-core systems mature, efforts like NOC may address the problem of cache coherence for which multi-core performance can suffer. The cache coherence problem in a parallel multiprocessor environment arises when copies of a shared resource in memory are maintained in local caches and modified without maintaining the consistency between the cache and memory [18]. This problem is amplified in multi-core and many-core systems where the sharing of the cache between cores introduces an intervening level of *spatial* complexity between local caches and the multi-core processor memory. *Temporal* locality of cache consistency also suffers.

Sorting out the performance issues of a generic multi-core processor system is left in the hands of the application's developer. The details of achieving optimal performance on a dual- or multi-core system is, in some sense, an extension of code development for traditional single-core multiprocessors. One difference arises from the hierarchical cache organization of multi-core systems. This one modification can skew the *spatial* and *temporal* locality of references depending on the multithreaded demands of the application. The skewing of references may result in positive or negative performance behaviors. However, it is a new programming or thread scheduling issue that must be confronted by the code developer. The art of multi-core programming is to devise multithreaded scheduling or throttling schemes to deal with the interactions between shared cores and the shared L2 cache resource. As the multi-core processor industry matures, manufacturers and their designers may provide better hardware support to facilitate the programmer's tasks. Until that time, it is the struggles of the programmer in extracting the last ounce of performance, and in so doing provide examples that guides the industry in developing new and better architectures.

4.2. GPUS

In this section, the architecture of the Nvidia GeForce GTX 280 is described as an example of a generic GPU architecture. The GTX 280 supports shared memory, atomic operations, double precision floating point instructions, 240 cores that run at 1.3GHz, and manufacturing that uses the 65 nm fabrication process. The Nvidia GPU is a streaming SIMD processor, whose kernel represent single program multiple data (SPMD) programming strategies. This interesting hybrid model is flexible but introduces constraints that must be understood to avoid performance degradations.

Fig. 7. is a high-level view of the Thread Processing Cluster (TPC). The TPC consists of 3 stream multiprocessors (SM) each with eight (8) streaming processors (SPs). Within a SM, each block of eight (8) SPs and raster operation processors (ROP) for graphics share an Instruction Unit (IU), and executes threads in a Single Instruction Multiple Data (SIMD) mode. Any diverging (branching, data stalling, etc.) threads may not execute in parallel. Local shared memory in each of the three SMs allows associated SPs to share data with other intra-SPs. The shared memory organization is essential to the performance of SPs. Shared memory is used to minimize the need to read or write to/or from the global memory subsystem (shown in Fig.8), which are typically very slow. Inside each TPC, eight texture-filtering units (TF) are provided for graphic rendering tasks. A typical memory hierarchy may have 32-bit registers per SP (see Fig.9). SMs also have access to constant and texture caches. Constant and texture caches are read-only and important to SMS performance since they can be accessed quickly.

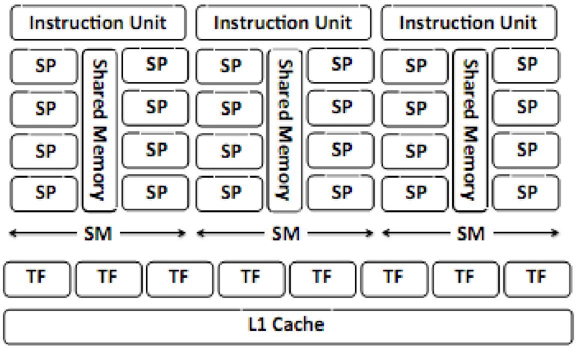


Fig. 7 – Thread Processing Cluster (TPC)

Fig. 8, depicts the GeForce GTX 280 architecture. The GeForce GTX 280 provides 10 TPCs, with three (3) SMs per TPC. The GPU employs synchronous multithreading. It has a hardware-based thread scheduler manages that schedules threads across the TPC. If a thread is stalled on a memory access, the scheduler can context switch to another thread with little overhead. A typical GPU has upwards of 30,720 threads on a chip. The architecture includes texture caches and memory interface units. The texture caches are used to combine memory accesses more efficiently and provide higher bandwidth memory read/write operations. The texture cache is a level 2 (L2) cache and is shared by all TPCs. The "atomic" operations perform atomic read-modify-write operations to global memory and facilitates parallel reductions and parallel data structure management. Atomic operations are integer operations in global memory that provide associative operations on signed/unsigned ints; add, sub, min, max, and, or, xor; increment, decrement; exchange; and compare and swap operations. Table 1 provides a summary of some important parameter values that define the Nvidia GTX 200 series [19,20].

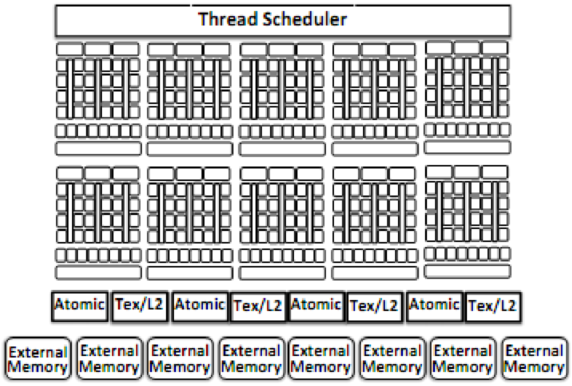


Fig. 8 – GeForce GTX 280

Table 1.– GPU Specifications

SM Resources	
SP	8 per SM
Registers	16,384 per SM
Shared Memory	16KB per SM
Caches	
Constant Cache	8KB per SM
Texture Cache	6 – 8KB per SM
Shared Memory	16KB per SM
Programming Model	
Wraps	32 threads
Blocks	512 threads max
Registers	128 per thread max
Memory	8 ×128MB, 64-bit
Constant Memory	64 KB total

CUDA is the synonym for the Compute Unified Device Architecture. It is a software specification-programming platform to assist the GPU program developer to interface to the logical GPU processor. Using CUDA, the programmer can specify the thread layout that are organized in blocks and in turn organized into grids. Threads executing within a block can synchronize among themselves. Thread communication between different blocks must be performed through slower global memory. Threads in a block have IDs and can be indexed into 1, 2, or 3 dimensions. A grid consists of multiple thread blocks of the same dimension and size. All threads in a grid execute the same CUDA kernel (SPMD). The CUDA architecture allows a SM to execute blocks one at a time. A "warp" of 32 threads or a half warp of 16 threads defines the number of threads that can execute in parallel. Each block is executed in 32 thread Warps. Warps are the scheduling units for the SM; and thus, thread scheduling is measured in warps. A block size can be defined to be 1 to 512 concurrent threads. At any time, only one warp is executed by a SM. Warps whose next instruction has its operands ready for execution are placed into the ready queue. Ready warps are selected for execution on a prioritized scheduling policy. All threads in a warp execute the same instruction (SIMD) when selected. The hardware is free to assign blocks to any SM at any time. Each block can execute in any order relative to other blocks. Threads in the same block share data and synchronize while doing their portion of the work. Threads in different blocks cannot cooperate.

Fig. 9, illustrates a CUDA memory architecture of the GPU. Global memory (off-chip) is the main means of communicating R/W data between host and memory. Content and Texture memories (off-chip) are available to all threads but when cached, their access latency is very short. Shared memory (on-chip) is shared between threads in the same block. Shared memory is divided into "memory banks" of equal size and is as fast as register when no bank conflicts occur. The amount of shared memory per SM is 16 KB organized into 16 banks.

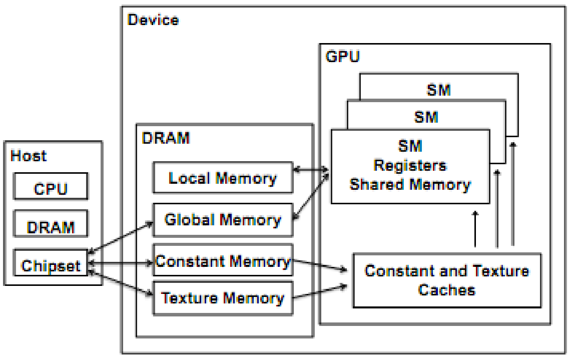


Fig. 9 – CUDA Memory Model

Since the wrap size is 32 threads, a half wrap size of 16 is used for conflict-free bank memory access. Each thread has a private local memory (off-chip). Local memory is to replace the functionality of registers if the kernel requires more registers than are available to a thread. Load/store instructions swap register memory loads and stores in and out of the local memory, which slows down the program performance. CUDA threads may access data from multiple memory spaces during their execution. Finally, all threads have access to the same global memory. Registers are very fast and play a significant role in performance tuning.

The CUDA SPs support lightweight threads designed to execute in SIMD mode. CUDA cores lack their own register files or L1 caches. They do not have multiple function units for each data type (floating point and integer) or load/store unit for accessing memory. Each CUDA core has a pipelined floating-point unit (FPU), a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results.

CUDA SPs are optimized for multithreaded performance but not optimized for single-threaded performance [21].

GPUs provide fine-grain multithreading to offset the extreme latencies incurred from slow main memory accesses. The characteristics of fine-grain multithreading require very high thread utilization [22,23]. Multithreading for the GPU requires explicit optimization of the number of registers used by the kernel and the number of threads per block. The SIMD nature of each kernel execution, dictates that simpler and smaller kernels (that are written in SPMD fashion) can provide better performance. Smaller computational kernels require less registers and are less likely to have data stalls, branching or frequent communications. GPU kernels must be organized and scheduled to maintain a balance between *temporal* and *spatial* locality. Application programs that exhibit regularity in their computational structure are, therefore, prime candidates to profit from the use of GPUs as general-purpose applications on GPUs (GPGPU). The program developers have the primary responsibility to perform microsurgery on applications for implementation on the GPU. The programming task is arduous but can be rewarding. The GPU program developers must remind themselves that fine-grained SIMD processing can return high performance gains but only if there are enough resources and allows only limited communications. Of course not all program applications can support the number of threads required to sustain high GPU performance. As a consequence, the GPU has emerged as a co-processor to the CPU [24].

In summary, the GPU architecture and the multithreading interactions can be a very complicated and dynamic environment for which to design optimal programs. Multithreading, if poorly coordinated, can result in large latencies in operand accesses and contenting thread delays. The design of efficient GPU programs requires assessing the latencies within the components of the memory hierarchy; and uses this knowledge to develop a mapping of multithreading strategies that accrue benefits from the aggregation of data utilization according to regular access patterns. Ultimately, it is the *spatial* and *temporal* localities that include read/write accesses, speeds of the different memory modules, and the threaded structure of the kernel's execution blocks that dictates the performance that can be achieved.

5. CONCLUSION

Assessing the programming issues of modern processors appears to be a confusing and challenging task. This presentation argues that by employing two simple principles of an expanded notion of *spatial* and *temporal* locality some of the most intricate pitfalls and traps can quickly be exposed and reasoned about. This is not to imply that the programming techniques will be easy to derive but rather that performance issues can be isolated and dealt with by using a more rational algorithmic approach. In multi-core systems, the sharing of the cache between several cores has the unintended downside of incurring higher data access congestion between cores. In addition, cache coherence also becomes a performance consideration over which the programmer has some degree of explicit control. In future evolutions of multi-core systems, new hardware and software technologies may mitigate or solve these problems.

The massively parallel programming paradigm of the GPU exemplifies the code developer's dilemma in fully understanding the organization of the system's architecture. Before attempting to write a single line of program code, the GPU's *spatial* and *temporal* organization should be studied in considerable detail. The optimal performance of a multithreaded GPU architecture demands that all threads be active at any one time and that they are all making uniform progress. This is a challenge that requires practice and knowledge based upon trial and error. GPU programming practices can be gained by learning techniques (tricks) from experienced GPU program developers; however, it is more important to learn how to design one's own programming strategies, as modern computer systems will definitely evolve away from current technologies and hardware organizations.

The art of high performance programming is a combination of first principles taken as a foundation to unravel the bottlenecks and latencies associated with the architectures of different systems. Equipped with this information, the programmer must articulate appropriate coding techniques, task-scheduling schemes, and resource allocations to mitigate or eliminate perceived latencies and bottlenecks within the framework of a given architecture.

6. REFERENCES

[1] P.J. Denning, S.C. Schwartz, Properties of the working-set model, *Communications of the ACM*, (15) 3 (1972), pp. 191-198.

[2] P.J. Denning, The locality principle, *Communications of the ACM*, (48) 7 (2005), pp. 19-24.

[3] M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison Wesley, June 16, 1995.

[4] J.L. Gustafson, Reevaluating Amdahl's law, *Communications of the ACM*, (31) 5 (1988), pp. 532-533.

[5] G. Amdahl, Validity of the single processor approach to achieving large-scale computing capabilities, *AFIPS Conference Proceedings*, (30) (1967), pp. 483-485.

[6] D.H. Bailey, Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers, *RNR Technical Report* RNR-91-020, June 11, 1991.

[7] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, and R. Biswas, scientific application-based performance comparison of SGI Altix 4700, IBM POWER5+, and SGI ICE 8200 Supercomputers, *Proceedings of the ACM/IEEE conference on Supercomputing, SC'08*, 2008.

[8] S. Hily and A. Seznec, Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading, *Technical Report* PI-1086, IRISA, 1997.

[9] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S.W. Keckler, A NUCA substrate for flexible CMP cache sharing, *Proceedings of the 19th annual international conference on Supercomputing ICS'05*, 2005, pp. 31-40.

[10] W.-D. Weber and A. Gupta, Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, June 1989, pp. 273-280.

[11] B.J. Smith, Architecture and applications of the HEP multiprocessor computer system, *SPIE*, (298) (1981), pp. 241-248.

[12] G.R. Andrews, Paradigms for process interaction in distributed programs, *ACM Computing Surveys*, 1991.

[13] P.E. McKenney and J. Slingwine, Efficient kernel memory allocation on shared-memory multiprocessors, *In USENIX Conference Proceedings*, Berkeley CA, February 1993.

[14] M.I. Reiman and P.E. Wright, Performance analysis of concurrent-read exclusive-write, *ACM*, (February 1991), pp. 168-177.

[15] J. Sartori and R. Kumar, Low-overhead, high-speed multi-core barrier synchronization, high performance embedded architectures and compilers, *Lecture Notes in Computer Science*, (5952) (2010), pp. 18-34.

[16] M. Amde, T. Felicijan, A. Efthymiou, D. Edwards, and L. Lavagno, Asynchronous on-chip networks, *IEE Proceedings Computers and Digital Techniques*, (152) 02 (2005).

[17] A.O. Balkan, M.N. Horak, G. Qu, U. Vishkin, Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing, *Proc. IEEE Symp. on High Performance Interconnection Networks* (Hot Interconnects), August 2007.

[18] P. Stenstrom, A survey of cache coherence schemes for multiprocessors, Computer, (23) 6 (1990), pp. 12-24.

[19] Nvidia, "Compute Unified Device Architecture Programming Guide Version 2.0," http://developer.download.nvidia.com/compute / cuda/2 0/docs/NVIDIA CUDA Programming Guide 2.0.pdf.

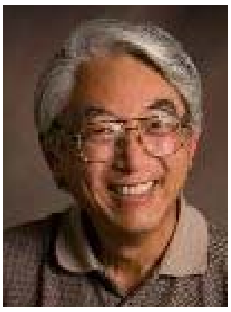
[20] Nvidia, "NVIDIA GeForce GTX 200 GPU Architectural Overview," http://www.nvidia. com/docs/IO/55506/GeForce GTX 200 GPU Technical Brief.pdf, May 2008.

[21] T. Halfhill, Looking Beyond Graphics NVIDIA's Next-Generation CUDA Compute and Graphics Architecture, http://www.nvidia. com/content/PDF/fermi_white_papers/T.Half hill_Looking_Beyond_Graphics.pdf

[22] A. Kumar, Tips for speeding up your algorithm in the CUDA programming, http://www. mindfiresolutions.com/Tips-for-speed-up-your-algorithm-in-the-CUDA-programming-399.php.

[23] N. Satish, M. Harris and M. Garland, Designing efficient sorting algorithms for manycore GPUs, *Proc. 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[24] A. Lippert, NVIDIA GPU Architecture for General Purpose Computing, http://www.cs.wm. edu/~kemper/cs654/slides/nvidia.pdf



Dr. Robert E. Hiromoto*, received his Ph.D. degree in Physics from University of Texas at Dallas. He is professor of computer science at the University of Idaho. His areas of research include information-based design of sequential and parallel algorithms, decryption techniques using set theoretic estimation, parallel graphics rendering algorithms for cluster-based systems, and secure wireless communication protocols.*